1. Как работает система сцены в Godot и в чем ее преимущества?

Приведите блок схему простенького платформера на Godot, где есть:

- *Стартовый экран загрузки с текстом*
- *Экран меню*
- *Игровой экран с персонажем и противников*

В годот система сцены построена вокруг нодов и их иерархии. Нод или узел — это базовый элемент, выполняющий конкретную функцию. Несколько нодов можно объединять в дерево, и такое дерево сохраняется в отдельный файл как сцена .tscn или .scn. Каждая сцена имеет один корневой узел и может содержать неограниченное количество дочерних. Сцены можно вкладывать друг в друга, повторно использовать и создавать их инстансы внутри других сцен.

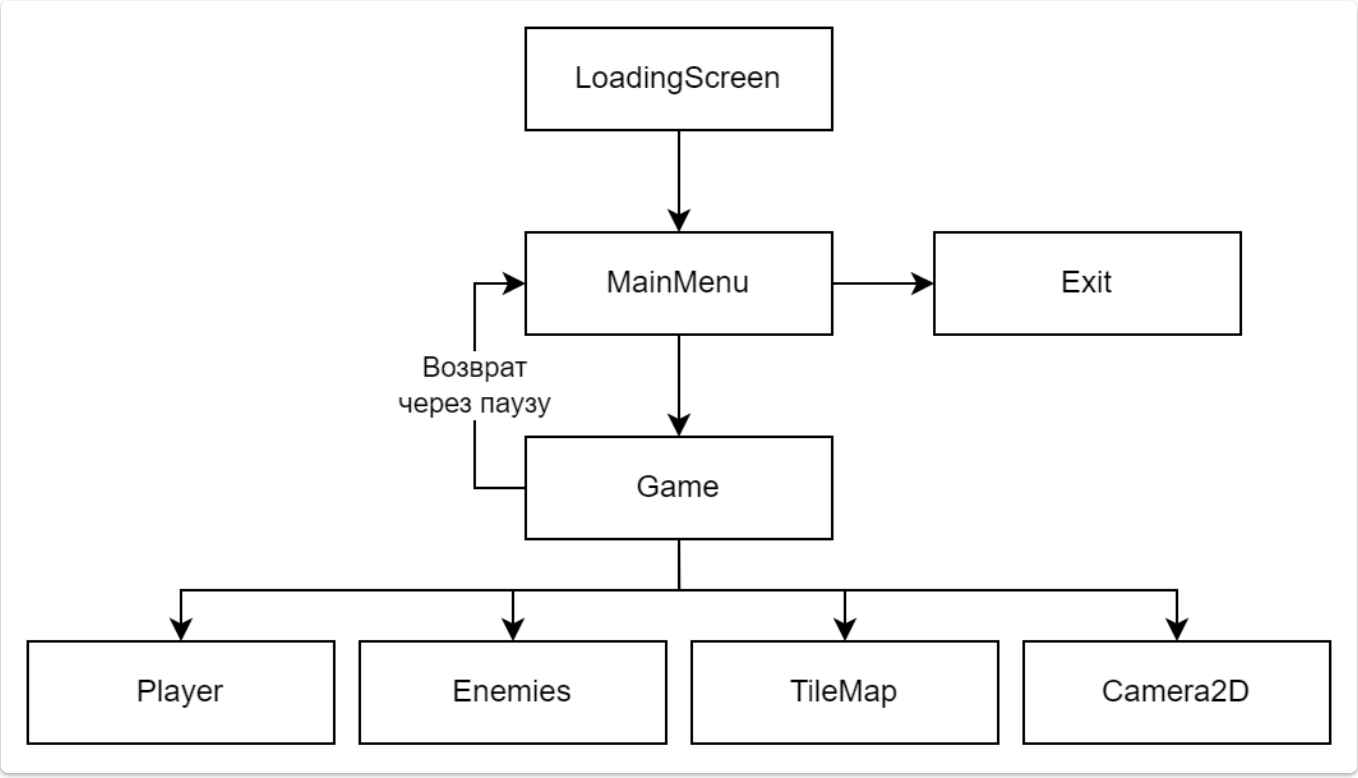
Главное преимущество системы сцены в заключается в её модульности и гибкости. Любой объект в игре реализуется как отдельная сцена, которую затем можно многократно использовать в других местах. Это упрощает разработку, поскольку изменения в исходной сцене автоматически отражаются во всех её экземплярах.

Кроме того, сцены поддерживают наследование: можно создать базовую сцену, например для врага, и на её основе создавать разные вариации с уникальными свойствами.

Пример:

```
project/
├── LoadingScreen.tscn
│   └── Label ("Loading...")
│
├── MainMenu.tscn
│   ├── VBoxContainer
│   │   ├── Button ("Start game")
│   │   └── Button ("Exit")
│
├── Game.tscn
│   ├── Player (CharacterBody2D)
│   │   └── Sprite2D
│   │   └── CollisionShape2D
│   ├── Enemy (CharacterBody2D)
│   │   └── Sprite2D
│   │   └── CollisionShape2D
```

```
|   ├── TileMap
|   └── Camera2D
```



2. Объясните, как работают сигналы (Signals) в Godot?

Приведите общий пример использования сигналов и\или двух несвязанных между собой node (Не имеющие общего родителя)

Сигналы - это встроенный инструмент делегации годот, позволяющий взаимодействовать одному игровому объекту с другим без прямых зависимотей. Один объект "слушает" сигнал, второй объект "испоускает" его.

Сигналы могу устанавливаться как через инспектор, но в этом случае это встроенные готовые сигнаалы каких-либо объектов, так и через код, где можно создавать свои кастомные.

```gdscript
# кастомный сигнал
# объявление сигнала
signal cuatom_signal(value)

# испускание сигнала
emit_signal("custom_signal", value)
custom_signal.emit(value)

# подключение сигнала
some_object.connect("custom_signal", "_on_custom_signal")
```

```
custom_signal.connect(_on_custom_signal)

# целевой метод
func _on_custom_signal(value):
    print(value)
```

Пример:

Player - собирает монетки

```
signal coin_collect(value)

func _on_coin_area_collide(coin):
    coin_collect.emit(coin.value)
```

UI - отображение собранных монет

```
func _ready() -> void:
    # находим и присоединяем сигнал player'a
    var player = get_tree().get_first_node_in_group("player")
    player.coin_collect.connect(_on_coin_collect)

# обновляем счётчик label
func _on_coin_collect(value):
    coins.text = str(int(coins.text) + value)
```

3.Как в GDScript организовать наследование, и зачем это нужно?

Наследование — это механизм, позволяющий создавать новые классы на основе существующих, перенимая их свойства и поведение, а также добавляя или изменяя функциональность.
Наследование делает код более организованным, уменьшает дублирование и облегчает поддержку проекта по мере его роста.

Таким образом можно создать один общий класс, который содержит базовую функциональность, а затем создавать специализированные классы-наследники, которые расширяют или изменяют эту функциональность без дублирования кода.

Пример

базовый класс Оружия:

```
extends Node
class_name Weapon
```

```
var damage : int = 10
var attack_speed : float = 1.0
var durability : int 10

func attack():
    print("Dmg done: ", damage)
```

Наследование - новый тип оружия Меч

```
extends Weapon
class_name Sword

func _init():
    damage = 15
    attack_speed = 0.7

# переопределение метода атаки
func attack():
    print("Using sword")
    # вызов родительского метода
    .attack()

# добавление функионала
func big_attack():
    print('Big swing')
    damage *= 2
    .attack()
    damage /= 2
```

4. Как работает система импорта ресурсов в Godot? Что произойдет, если изменить исходный файл изображения?

Система импорта в Godot преобразует внешние файлы в оптимизированные форматы, которые движок может эффективно использовать.

Годот отслеживает папку проекта и автоматически добавляет новые файлы. При добавлении новых файлов создаются файлы .import с различными настройками импорта.

При изменении файла происходит автоматическое обновление. Так же идёт обнаружение изменений в папке пректа и идёт автоматический переимпорт. После чего идёт обновления ссылок, в которых все ноды и ресурсы, использующие этот файл, автоматически обновляются.

Если изменяется файл изображения, допустим, в графическом редакторе, то годот автоматически переимпортирует .png в формат .stex. Спрайт мгновенно показывает обновленное изображение. И в этом случае не требуется перезагрузка проекта или ручное обновление

5. Что такое _process() и _physics_process() в GDScript и чем они отличаются?

Функции _process() и _physics_process() - это встроенные функции, которые автоматически вызываются движком для каждого узла, если они переопределены. Они используются для обновления состояния игры, но в разных контекстах.

Основное отличие _process() и _physics_process() в том, что первый вызывается каждый кадр, а второй фиксмрованное установленное количество раз.

```
_process()
```

- вызывается каждый кадр рендеринга, его частота зависит от фпс и может меняться
- предназначен для логики игры, анимаций, интерфейса и обработки ввода
- не синхронизирован с физическим движком
- delta переменное значение, время с последнего кадра
- может вызываться реже при низком фпс, что влияет на плавность
- обычно используется для визуальных эффектов, интерфейса и игровой логики

```
_physics_process()
```

- вызывается с фиксированной частотой, которое изменяется в настройках проекта
- предназначен для физики, движения, коллизий и всего связанного с физическим движком
- синхронизирован с физическим движком Godot
- delta фиксированное значение, обычно 1/60 секунды
- гарантирует стабильную работу физики независимо от фпс
- обычно используется для перемещения объектов, проверки столкновений и для физики

6. Как создать и использовать таймер (Timer) в Godot?

Таймер можно создать двумя способами: с помощью редактора, подключив ноду Timer, или через код, использовав класс Timer.

Пример создания в коде:

```gdscript
func _ready():
    # объявление таймера
    var timer : Timer = Timer.new()
    add_child(timer)

    # настройка параметров
    timer.wait_time = 2.0
    timer.one_shot = false

    # подключение сигнала таймаута таймера
    timer.timeout.connect(_on_timer_timeout)

    # стартуем
    timer.start()
```

Альтернативный вариант:

```gdscript
get_tree().create_timer(1.0)
```

7. Объясните, как работает система слоев и масок (Layers and Masks) для коллизий в Godot.

Система слоев и масок в использует битовые маски для управления взаимодействиями между объектами.
Здесь используется 32 бита в 2д или 64-бита в 3д, где каждый бит представляет отдельный слой.

- Слои определяют, к какой группе принадлежит объект
- Маски - определяют, с какими группами объект может взаимодействовать
  При этом каждый объект может принадлежать сразу нескольким слоям.

В проекте коллизии в основном настраиваются через инспектор, где в настройках можно задать имя для читаемости каждому биту.

Коллизия происходит, если есть пересечение между слоями одного объекта и маской другого, и наоборот.

То есть, объект A сталкивается с объектом B, если:

```gdscript
(A.collision_layer & B.collision_mask) != 0 & (B.collision_layer &
A.collision_mask) != 0
```

Но чаще всго используется односторонння коллизия как например для Area2D (пример выше был для физических объектов):

```
(other_object.layer & area.mask) != 0
```

Также можно использовать динамическое изменение слоев коллизии через код:

```
# добавить 3 слой
collision_layer |= 1 << 2

# убрать 3 слой
collision_layer &= ~(1 << 2)

# инвертировать 3 слой
collision_layer ^= 1 << 2
```

8. Как в GDScript организовать взаимодействие между разными сценами или узлами?

Основные способы взаимодействия между разными нодами:

- сигналы - механизм событий, позволяющий нодам уведомлять другие ноды о происходящем
- группы - позволяют группировать ноды и выполнять над ними операции или отправлять сообщения

  ```
  # добавление в группу
  add_to_group("group")

  # вызов метода у всех узлов группы
  get_tree().call_group("group", "do_smth")

  # получение всех нодоов группы
  var in_group = get_tree().get_nodes_in_group("group")
  ```

- ссылки на узлы - получение ссылки на над через путь в дереве сцен

  ```
  # абсолютный путь
  var player = get_node("/root/Main/Player")

  # относительный путь
  var parent = get_parent()
  var sibling = get_node("../SiblingNode")

  # использование $ синтаксиса
  ```

```
var child = $ChildNode
var nested_child = $Container/ChildNode

# использование уникального имени %
var child = %ChildNode
```

- автолоад - глобальные скрипты, которые доступны из любой сцены

  обычно это какой-нибудь GameManager

```
var score = 0
signal score_changed

func add_points(points):
    score += points
    score_changed.emit(score)

# ----------------------------

# любая сцена может обращаться
func _ready():
    GameManager.score_changed.connect(_on_score_changed)
    GameManager.add_points(100)
```

9. Как загрузить и инстанцировать сцену динамически во время выполнения игры?

Основные способы загрузки сцены это load(), preload() и instantiate()

```
# загрузка сцены во время выполнения
var scene_resource = load(res://scenes/scene.tscn)
# предзагрузка при звгрузки скрипта
var scene_resource = preload("res://scenes/scene.tscn")
# создание инстанса
var scene_instance = scene_resource.instantiate()
# добавление инстанса на сцену
add_child(scene_instance)
```

10. Какие средства профилирования и отладки предоставляет Godot? Как ими пользоваться?

В годоте есть втроенные инструменты отладки такие как:

- дебаггер, который показывает ошибки, предупреждения и логи
- визуальный отладчик, позволяющий включить видимые формы столкновений,

навигационную сетку, и включается через меню отладки

- точки останова
  - имеются как встроенные в редактор, так и их вызов через код

```
# пример условной точки останова
func _process(delta):
    # есловный вывод только при определенных условиях
    if player.health < 10 and Engine.is_editor_hint():
        # точка останова
        breakpoint
        print("low hp")
```

- логирование
- великий print()

Втроенные интсрументы профилирования:

- перфоманс монитор
  - показывает фпс, использование памяти и цпу
  - можно включить через дебаггер при запущенном проекте соответсвенно
- профайлер
  - записывает выполнение функций и затраченное на них время
  - можно включить через дебаггер-перфоманс
- также имеются инструменты профилирования встроенные в код

  ```
      var fps = Performance.get_monitor(Performance.TIME_FPS)
      var memory = Performance.get_monitor(Performance.MEMORY_STATIC)
  ```

  При этом основной профилировщик не запускается автоматически, т.к. профилирование требовательно к производительности. Он должен постоянно измерять всё происходящее в игре и отправлять отладчику данные, поэтому по умолчанию отключен.

Чтобы запустить профилирование, нужно запустить игру, а затем в редакторе нажать кнопку «Start» в левом верхнем углу вкладки

При этом всё же можно установить автозапуск, чтобы профилировщик автоматически запускался при следующем запуске проекта. Но состояние автозапуска не сохраняется между сеансами редактора

С его помощью можно находить неоптимизированные участки кода, наблюдая за графиком производительности и отслеживая скачки

11. Как реализовать систему сохранения и загрузки данных игры в Godot? Какие существуют подходы и какие классы для этого используются?

Поддерживаются сохранения в такие текстовые форматы как JSON, ConfigFile, Resource

- так json универсальный и удобный для сериализации сложных структур, в основном используется для сохранения игровых данных типа квестов и инвенторя

```
var save_path = "user://savegame.json"

# пример сохранения
func save_game(data: Dictionary) -> void:
    var file = FileAccess.open(save_path, FileAccess.WRITE)
    if file:
        var json_str = JSON.stringify(data)
        file.store_line(json_str)
        file.close()

# пример загрузки
func load_game() -> Dictionary:
    if not FileAccess.file_exists(save_path):
        return {}
    var file = FileAccess.open(save_path, FileAccess.READ)
    var content = file.get_as_text()
    file.close()

    var result = JSON.parse_string(content)
    if result == null:
        return {}
    return result
```

- конфиг — встроенный формат .cfg обычно испльзуется для небольших настроек и опций

```
# пример сохранения
func save_settings():
    var config = ConfigFile.new()
    config.set_value("player", "health", 100)
    config.set_value("player", "position", Vector2(50, 100))
    config.save("user://settings.cfg")

# пример загрузки
func load_settings():
    var config = ConfigFile.new()
    var err = config.load("user://settings.cfg")
    if err == OK:
        var health = config.get_value("player", "health", 100)
        var pos = config.get_value("player", "position", Vector2.ZERO)
```

```
        print(health, pos)
```

- можно сохранять данные прямо в ресурс, в годоте обычно используются форматы .tres или .res

```
# создание кастомного ресурса для сохранения данных игрока
extends Resource
class_name SaveData

@export var player_name : String = "player"
@export var health : int = 100
@export var position : Vector2 = Vector2(0, 0)

# ----------------------------------------

# пример функции сохранения
func save_game():
    var data : = SaveData.new()
    data.player_name = "player 1"
    data.health = 75
    data.position = Vector2(10, 20)

    # сохраняем как читаемый .tres
    ResourceSaver.save(data, "user://savegame.tres")

    # сохраняем в бинарный формат .res
    ResourceSaver.save(data, "user://savegame.res")

# пример функции загрузки
func load_game():
    if ResourceLoader.exists("user://savegame.tres"):
        var data: SaveData = ResourceLoader.load("user://savegame.tres")
        print("name:", data.player_name)
        print("HP:", data.health)
        print("position:", data.position)
```

Так же имеется возможность использовать бинарное сохранение FileAccess + store_var / get_var

Можно реализовать автосохранение через Node._get_property_list() и Node.get(), что позволит автоматически выгружать свойства нодов и затем их восстанавливать. Это можно использовать для сохранения состояния сцен

Также есть ещё внешний аддон Save/Load Manager

Соответсвенно основные классы для сохранения:

- FileAccess - открывает, читает и пишет файлы
- DirAccess - создание папок для сохранений
- ConfigFile - позволяет сохранять и загружать данные в .ini:
- ResourceSaver / ResourceLoader - сохраняет данные в .tres или .res

12. Как подключить и использовать Android плагины в Godot? Какие шаги необходимы для интеграции?

Для начала необходимо:

- установить OpenJDK
- установить AAAAAAndroid Studio
- настроить расположение Android SDK

Далее надо создать внутри проекта папку android, содержащую все необходимые файлы среды выполнения. Создаётся это автоматически в редакторе годота, через меню проекта.

Далее самый простой способ установить внешний плагин через меню библиотеки ассетов годота

Далее настраиваются отдельные шаблоны экспорта для каждого устройства, так как для каждого устройства потребуется собственный загрузчик. Это делается в меню экспорта.